Preliminary Design of the SAFE Platform

Preliminary Design of the SAFE Platform

André DeHon* Ben Karel* Thomas F. Knight, Jr.† Gregory Malecha‡
Benoît Montagu* Robin Morisset§ Greg Morrisett‡ Benjamin C. Pierce*
Randy Pollack‡ Sumit Ray† Olin Shivers¶ Jonathan M. Smith* Gregory Sullivan†

ABSTRACT

SAFE is a clean-slate design for a secure host architecture. It integrates advances in programming languages, operating systems, and hardware and incorporates formal methods at every step. Though the project is still at an early stage, we have assembled a set of basic architectural choices that we believe will yield a high-assurance system. We sketch the current state of the design and discuss several of these choices.

1. INTRODUCTION

Computer systems remain distressingly insecure. The full set of reasons is a matter of debate, but one factor that certainly makes progress more difficult is the numerous design decisions that were made decades ago and are now deeply embedded in the hardware and software ecosystem. We feel the time is ripe to consider an integrated redesign of the entire system stack, from hardware to applications, with an eye to simplicity and security throughout.

Our effort is based on two fundamental insights. The first is that formal methods—detailed, machine-checked proofs of critical properties—have now matured to the point where they can play a key role in the design of serious systems. In particular, we know how to write formal specifications of the semantics of full-blown instruction sets and of the higher-level abstractions provided by high-level programming languages and software services, and we can build machine-checkable proofs that each layer is correctly implemented by the software running on the layers beneath it. These proofs require significant effort, but making this effort an integral part of the design process has the salutary effect of exerting significant pressure to streamline all aspects of the design. The second insight is that recent years have seen a rapid increase in hardware resources. We can now afford to reconsider some of the traditional sources of complexity in operating systems—e.g., virtual memory, complex interrupt handling schemes, and manual storage allocation and deallocation—and use simpler designs for which proofs are more tractable. Moreover, we can spend hardware to efficiently enforce desirable policies at the lowest level of the system—e.g., word-level information flow tracking and fine-grained, least-privilege protection domains with cheap domain crossing. The resulting system will not be suitable for all purposes—it will use more energy and run slower than systems optimized for high performance or low power with no regard to safety—but, given the value of information on today's systems and the hostile networked environment in which they must operate, such concessions for security are now warranted for many systems.

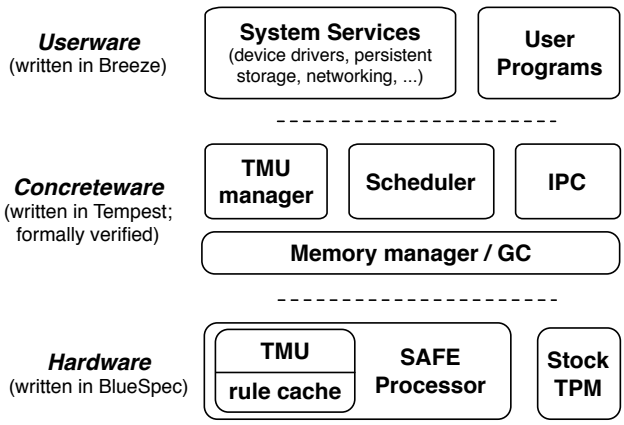

Figure 1: SAFE Architecture

This paper describes the SAFE design as it currently exists, nine months into the project, discusses some of its more interesting and potentially controversial choices, and sketches remaining challenges. (Because SAFE spans the range of system layers, the related literature is vast; to conserve space in this short overview, we cite and discuss just a few points of comparison.)

2. ARCHITECTURE OVERVIEW

The architecture of the SAFE platform, sketched in Figure 1, comprises three distinct layers. The *hardware* layer consists of a microprocessor for a novel instruction set supporting object-level type safety with unforgeable pointers (capabilities), hardware-enforced distinctions between pointers,

*University of Pennsylvania
†BAE Systems
‡Harvard University
§École Normale Supérieure Paris
¶Northeastern University

PLOS '11, October 23, 2011, Cascais, Portugal.

data, and instructions, and information-flow tracking via fine-grained tagging. In our tagging scheme, every word in memory and registers includes both a data payload and a large (pointer-sized) tag which encodes information about its type, provenance, and access limitations. These tags are interpreted by a Tag Management Unit (TMU) that—on every instruction, in parallel with the processor's main data path—looks up the tags on the current instruction, its operands, and the current program counter in a hardware *rule cache* to determine whether the instruction is permitted and, if so, how its results should be tagged. The hardware layer also contains a Trusted Platform Module (or a more general-purpose Hardware Security Module) for handling operations such as generating and storing cryptographic keys and performing (non-bulk) encryption.

Concreteware is a thin layer of software that wraps the facilities of the raw hardware into a clean abstract machine, on which the rest of the SAFE software rests. In particular, it provides allocation and garbage collection of memory frames (bounded regions named by unforgeable pointers), scheduling, and message-passing interprocess communication. Concreteware also implements a TMU Manager that is responsible for filling the hardware rule cache, handling rule cache misses, and generating TMU rules from high-level access and information-flow policies expressed in a small domain-specific rule language.

The *userware* layer includes the bulk of what are usually thought of as operating-system services (device drivers, facilities for persistent storage, networking stacks, etc.), as well as ordinary user programs.

Ideas from programming languages play a pervasive role in this design. The hardware and concreteware together provide a run-time system for Breeze, a high-level language in which all userware-layer software will be written. Moreover, the concreteware layer itself will be coded almost entirely in a minimal subset of Breeze, called Tempest, suitable for systems programming.

Another pervasive concern in the SAFE design is the application of formal, machine-checked verification. Despite recent successes such as seL4 [9], formal verification of whole operating systems still requires daunting amounts of effort. But the SAFE architecture offers two opportunities for large payoffs from significantly smaller amounts of work. First, in a clean-slate design, we can tune every feature to smooth the verification task; for example we omit virtual memory (see 4.2), which caused a significant part of the proof effort in seL4 [9]. Second, we plan to verify only a limited range of security-critical properties for a fairly narrow slice of the system—the Tempest compiler and the concreteware layer. These proofs will provide strong safety guarantees, such as memory isolation and dynamic type safety, for higher-level system components, which in turn will simplify programmers' informal reasoning about the security of their programs.

3. THREAT MODEL

Our assumptions about the attacker's abilities are different for the various layers of a SAFE system.

We assume a correctly implemented instruction set architecture (to limit the scope of our verification efforts) and the absence of hardware-layer tampering, either via supply chain attacks or via prolonged physical access. However, we do assume that the attacker controls the system's (local or network-attached) persistent storage media. We must therefore use encryption to ensure the confidentiality and integrity of stored data. Similarly, we assume that a pre-constructed concreteware image is loaded at boot time into read-only memory.

At the user level, we assume there *will* be malicious code within the system—that is, there will be many processes running on behalf of many different principals, and some of these processes may attempt to compromise the secrecy or integrity of information created or used by others. Also, we assume that any secrets that are directly exposed to such malicious code can potentially be exfiltrated from the system by some overt or covert means. (This is in contrast to much of the existing work on language-based security, which only controls communication of data influenced by secrets via "official" channels and ignores the possibility of covert channels.) Moreover, since we do not propose formal verification of user-level code, even if a principal trusts the good intentions of the author of some piece of code, they should assume that the code may contain bugs or insecurities.

The SAFE design focuses on single-host security; we do not consider attacks at the level of networking protocols.

Although the novel aspects of the SAFE design are mostly focused on threats to confidentiality and integrity, threats to availability (exhaustion of resources, etc.) are also a significant concern. We will apply standard mitigation techniques in this area.

4. DESIGN HIGHLIGHTS

4.1 Language design and information flow

Breeze is a type-safe, mostly-functional language, similar in spirit to ML, except that it tracks information flow to support both confidentiality and integrity. Currently, type-checking and information-flow tracking are dynamic mechanisms, that directly reflect the capabilities of the hardware. Ultimately, we plan to add a static type system that will help guide the task of programming and provide an extra layer of checking, in keeping with the standard security principle of defense-in-depth.

Breeze encourages programming in a mostly-functional style, for a number of reasons. First, garbage-collected, immutable data structures simplify reasoning about safety and security properties: once a property is established, we don't need to worry about the data structure changing and invalidating the property. Second, a mostly functional language simplifies the development of concurrent code by supporting sharing and minimizing the need for synchronization.

The SAFE system tracks information flow to enforce programmer-supplied constraints on which data values may be read in which parts of the system. Approaches to information-flow tracking can be split roughly into two categories: programming language-based techniques [13], which are generally fine-grained and static, and techniques used in operating systems (e.g., [10, 4]), which are coarse-grained and dynamic. In contrast, both Breeze and the SAFE hardware support dynamic enforcement of fine-grained information flow, as done in [2]: indeed, a low-level dynamic approach allows our platform to run programs that were not statically checked or compiled with our compiler. This has several advantages: it makes our attack model more realistic, removes the compiler from the TCB, and allows us to track information flow even in very low-level systems

code that must be written in assembly.

One of the challenges of building a new system based on information flow is the wide variety of specific information-flow mechanisms that have been explored in the literature; in particular, the form of the labels attached to data values, the "label model," varies widely. This is natural, since the label model is essentially a small domain-specific language for expressing low-level security policies, and we might expect these to vary from between applications and between application-level and systems-level code. To retain flexibility, we are working to define a *generalized label model*—that is, a common interface that can be instantiated with many of the concrete label models described in the literature—along roughly the same lines as HAILS [**?**]. Both Breeze and the SAFE hardware are parametric with respect to this interface. The instance that we have used most heavily so far is a variant of the Myers-Liskov decentralized label model [13].

Many information-flow systems use *taint-tracking* mechanisms [17], which are only able to track *explicit flows*—flows that involve direct copying of sensitive information from one data structure to another. Others, including SAFE, also track *implicit flows* [8]—situations where the program's control state depends on secret data. A benefit of dealing with implicit flows is that we get a crisp statement of the security guarantee provided by the information-flow tracking mechanism: a *noninterference* theorem [6], stating (roughly) that the sensitive inputs of a program cannot influence its public outputs.

However, the noninterference theorem comes with some significant limitations. First, the termination-insensitive form (which is the one we are considering, termination-sensitive noninterference being much harder to check) is weaker in the presence of concurrent processes; for instance, a process can learn whether another process has died, perhaps as a result of testing some secret bit, by watching for (the absence of) side effects. Second, even in the single-threaded case, it only applies to release of information via channels that are captured by the formal operational semantics of the language; other channels—in particular, timing channels—are not excluded. These limitations mean that the simple story sometimes found in papers on information flow ("We allow the attacker's code to see and manipulate secret values, but that's OK because any data the attacker writes as a result will also be labeled, and this will ultimately prevent the attacker from exfiltrating it because we'll check the label at the point when it is about to get written over the network...") is dangerously misleading as a basis for building secure systems. Rather, the information-flow analysis must be supplemented with some form of access control mechanism, which programmers can use to prevent secret data from even being *seen* by untrusted code. Concretely, this is accomplished in SAFE by performing the access checks ("Does the authority of the current execution context include the right to read data tagged with this label?") not only at system boundaries, but at every point where the value of a secret might affect the program's internal behavior. These checks are performed by the TMU (Section 4.2) in parallel with the computation. For example, in order to add two integers, Alice must have access rights to both values.

In the Breeze design, we are exploring a number of ideas that are traditionally associated with capability-based systems [12], as well as ideas traditionally associated with access control. Our flexible information-flow framework and hardware support can express a wide range of possibilities, and we will exploit this to experiment with different mechanisms to contain and reason about information flow. For example, fine-grained capability passing makes it easy to dynamically create subsytems and assign them least-privilege access, while access control can be embedded in labels that are attached to values, and makes it possible to define *end-to-end* properties, absolutely limiting where a capability may flow.

Two major problems with information flow remain to be addressed. Declassification is required for any realistic system and obviously breaks noninterference if allowed without restraints: its interaction with end-to-end information-flow policies is a research challenge. And, as mentioned above, unrestricted concurrency is problematic, providing means for exploiting termination channels.

4.2 Hardware structures

The SAFE hardware has roots in architectures such as the Lisp machine [1]. Its unconventional structure is a result of shifting and refining the boundaries between operating system, language run-time and hardware through a co-design process; this recalls the Cambridge CAP and CMU Hydra/C.mmp efforts as described by Levy [12].

Our hardware architecture provides *tagged memory* and high-level abstractions such as capabilities, principals, and direct support for first-class functions. In contrast to (and informed by) earlier attempts such as the ill-fated Intel i432 [7], we believe the abundance of hardware today allows us to provide these advanced features without compromising performance. We include a generous register set and L1 cache, perform tag checking on hardware in parallel with operation, and we believe we can make authority-changing procedure calls as fast as conventional procedure calls. These improvements should eliminate the key sources of measured performance overhead in the i432 [5].

In the SAFE architecture, every word is associated with a tag. Unlike previous tagged architectures which used a small number of bits and a fixed interpretation, we use pointer-sized tags so that we can associate an arbitrary data structure with a value. The meaning of a tag is not fixed in advance; instead, programmable rules cached in the hardware Tag Management Unit specify the meaning of tags. This genericity permits tags to be used for a wide variety of purposes—as types, capabilities, information flow labels, access control specifiers, etc. In parallel with instruction execution, the processor routes the tags of the operands (including the PC) to the TMU to validate that the current authority has sufficient privileges to read the values of the operands and to execute the current instruction on them. At the same time, the TMU computes what tags should be placed on the resulting value and on the program counter. This is another example of spending plentiful hardware; in the common case where the rule is in the cache, the security check adds no time to the computation. This genericity should allow us to explore a wide range of fine-grained type- and security-policies for both systems and applications. Current challenges include designing high-level notations that compile down to TMU rules and developing effective programming idioms with small enough working sets to fit in the TMU's hardware rule cache.

Pointers to memory frames are provided as abstract, opaque data structures. Arbitrary pointers cannot be cre-

ated by anyone except a privileged memory allocation authority in the concreteware; in this respect they resemble *abstract types* in programming languages and *capabilities* in operating systems. Moreover, all pointers are *fat pointers* [3] holding base and bounds metadata, which permits safe pointer dereferencing by performing bounds checks directly in the hardware.

Two other essential features of the hardware are *authorities* and *gates*. An authority is a name associated with a set of capabilities and resources that can be used when performing a set of instructions. A gate is a low-level representation of a function that closes over an environment and an authority, similar to a gate in Multics [14]; a *gate call* is used to invoke the function, at the same time switching to the authority under which the gate was created. Possession of a pointer to a gate serves as an object-level capability for performing some action with an authority other than your own. For example, Alice may pass a gate to an untrusted principal Bob that, when executed, encrypts and declassifies data under Alice's authority. Thus, Bob gains a limited capability for declassification, but protected by encryption and under Alice's control. Hardware support for gates with TMU mediation allows domain crossing to be as inexpensive as a procedure call, removing one of the classic performance impediments to fine-grained compartmentalization.

The basic ALU and memory operations are generic RISC instructions. The novelty lies in how they are mediated by the hardware tags, authorities, and rules. We are also exploring native hardware support for threads, timer management, communication channels, and limited transactions.

4.3 Concreteware and system software

Because concreteware is subject to formal verification, many design decisions for the system-level components aim to simplify key interfaces such as scheduling, IPC, process isolation, and persistent storage.

Another fundamental concreteware service is interprocess communication via unidirectional, order-preserving channels. A desirable future improvement will be to enhance channel performance by integrating hardware-accelerated message passing. Hardware-accelerated frame copying, a SAFE version of DMA, is another attractive direction.

As a further simplification, the current SAFE design does not rely on virtual memory for process isolation. Instead, because pointers are not forgeable, it is easy to provide isolation between mutually suspicious processes running in a single address space by limiting access to sensitive references—a core idea of traditional capability systems [12].

The initial persistent storage model for SAFE is an object store, rather than a traditional file system. We have tentatively adopted a model reminiscent of the Hydra Object Storage System [15] system, in which every object exists either in memory or as a persisted "passive" version. In accordance with the threat model in Section 3, passivization requires encryption to ensure confidentiality and integrity.

Many fundamental questions remain open about how to structure the operating system. For example, meta-level operations such as debugging and logging are especially sensitive because they could be used to undermine information flow. A systematic design for secure and robust error handling remains to be explored, as does secure introspection of process status. Finally, separation of privilege and minimization of shared state naturally pushes the OS design towards that of a distributed system; we are still exploring the full impact of this line of thought.

5. ATTACKS

To illustrate the mechanisms we have described, we briefly consider several sorts of attacks (following the terminology of the Mitre Common Weakness Enumeration database) and sketch how each is addressed by SAFE.

Buffer-overflow attacks—and more generally, object and control-flow integrity violations—are completely prevented by object-level type safety, which is dynamically enforced through a combination of pointer bounds and tag checking.

Data-leakage attacks are addressed with a combination of information-flow tracking (for automatically maintaining connections between data values and their associated usage policies) and access control checks (for limiting the flow of sensitive information to untrusted code, which may try to leak it over covert channels).

SQL injections can also be avoided using the primitive information-flow tracking facilities of the platform—using integrity taints, for example, to check that SQL queries have been sanitized before execution.

Bypassing authorization checks is rendered difficult by having the hardware directly implement the necessary checks. The question of misconfiguration of privileges remains challenging; however, fine-grained decomposition of privileges minimizes the damage of any individual misconfigured check.

Hijacking privileged processes is addressed in multiple layers of the system. Least-privilege design helps minimize the effects of breaches; end-to-end information-flow tracking allows potentially malicious low-integrity inputs to be identified and treated with greater care; and type safety together with read-only code segments prevents code injection, return-to-libc, and other forms of control-flow hijacking.

Exploitable race conditions are a common, difficult problem with concurrent systems. There is no silver bullet for this class of vulnerabilities, but we can eliminate many low-level race conditions by allowing inter-process communication only via message passing, not shared mutable memory.

6. THE ROLE OF VERIFICATION

The design space for SAFE is large and full of subtle tradeoffs. In the face of this complexity, we believe that the use of formal techniques can significantly enhance the coherence of the design and increase confidence in the decisions we are making. (Of course, we also require experimentation and testing, which yield complementary insights.)

The specification of the SAFE instruction set architecture will be the project's most critical formal artifact, serving as the contract between the hardware and software sides of the system. It will be written as a program in the language of the Coq proof assistant, phrased in a low-level style that permits "extraction" into an executable symbolic simulator for validation of the hardware. (Formal verification of the hardware is beyond our current scope.)

We hope to formally verify the Tempest compiler, following the COMPCERT [11] verified C compiler; this will ease the burden of verifying the concreteware by allowing reasoning directly on its Tempest source. (In this respect we differ from the Verve project [16], where the low-level parts of the system, for which memory safety was verified,

were written in assembly.) We will attack the concreteware verification task by building and verifying a stack of increasingly abstract specifications on top of the Coq specification of the ISA—the first one abstracting away memory management, the second scheduling, etc.; the top of this stack will be a specification of the abstract machine presented by the concreteware to the rest of the software in the system.

7. STATUS

The majority of our effort so far has gone into the design and implementation of the Breeze language and the ISA, with accompanying formal specifications, interpreters and simulators. In particular, we have already defined a Coq semantics for the ISA, from which we can extract an executable simulator; we will use this simulator to cross-validate our FPGA-based implementation of the SAFE hardware. This brings us closer to an end-to-end functional system prototype.

We have also made progress on the formal verification side: we started verifying the correctness of a scheduler for a subset of the full ISA, and some theorems like noninterference for fragments of Breeze have been mechanically verified.

Our main current effort consists in developing user-level and system-level applications: this will not only allow us to validate the usability of the mechanisms that the SAFE system offers, but also permit us to start writing potential attacks against these programs to stress-test the protection mechanisms we are designing.

Acknowledgments. We thank Andreas Haeberlen and the anonymous reviewers for helpful comments on earlier drafts. This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

8. REFERENCES

[1] Bawden A, Greenblatt RD, Holloway J, Knight TF, Moon D, and Weinreb D. Lisp machine. *Artificial Intelligence*, pages 343–373, 1979. Winston PH, (ed.), v. 2, MIT Press, Cambridge.

[2] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. *SIGPLAN Notices*, 44:20–31, December 2009.

[3] J. Brown, J.P. Grossman, A. Huang, and Jr. T. F. Knight. A capability representation with embedded address and nearly-exact object bounds. Technical Report 5, MIT AI Lab, April 2000. Aries Project.

[4] Winnie Cheng, Aaron Blankstein, James Cowling, Dorothy Curtis, Vicky Popic, Dan R. K. Ports, David Schultz, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. Submitted for publication.

[5] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. Performance effects of architectural complexity in the Intel 432. *ACM Trans. Comput. Syst.*, 6:296–339, August 1988.

[6] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[7] D. Johnson. The intel 432: A VLSI architecture for fault-tolerant computer systems. *Computer*, 17:40–48, August 1984.

[8] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS 2008)*, pages 56–70, 2008.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *ACM SOSP*, pages 207–220. ACM, 2009.

[10] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. K., and R. Morris. Information flow control for standard OS abstractions. In *ACM SOSP*, Stevenson, Washington, USA, October 2007.

[11] Xavier Leroy. Formal verification of a realistic compiler. *Comm. of the ACM*, 52(7):107–115, 2009.

[12] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[13] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9:410–442, October 2000.

[14] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[15] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[16] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of PLDI'2010*, Toronto, Ontario, Canada, June 2010.

[17] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *ACM SOSP*, Big Sky, MT, USA, October 2009.